Fixed-Point & Floating-Point Number Formats

CSC231—Assembly Language
Week #13

Dominique Thiébaut
dthiebaut@smith.edu
Reference

http://cs.smith.edu/dftwiki/index.php/
CSC231_An_Introduction_to_Fixed-__and_Floating-
Point_Numbers
public static void main(String[] args) {

    int n = 10;
    int k = -20;

    float x = 1.50;
    double y = 6.02e23;
}

public static void main(String[] args) {

    int n = 10;
    int k = -20;
    float x = 1.50;
    double y = 6.02e23;
}

public static void main(String[] args) {
    int n = 10;
    int k = -20;
    float x = 1.50;
    double y = 6.02e23;
}

Nasm knows what 1.5 is!
in memory, x is represented by

00111111 11000000 00000000 00000000
or 0x3FC00000

Nasm knows what 1.5 is!
• Fixed-Point Format

• Floating-Point Format
Fixed-Point Format

• Used in very few applications, but programmers know about it.

• Some micro controllers (e.g. Arduino Uno) do not have Floating Point Units (FPU), and must rely on libraries to perform Floating Point operations (VERY SLOW)

• Can be used when storage is at a premium (can use small quantity of bits to represent a real number)
Review Decimal System

\[ 123.45 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} \]

Decimal Point
Can we do the same in binary?

- Let's do it with **unsigned numbers** first:

$$1101.11 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

Binary Point
Can we do the same in binary?

- Let's do it with **unsigned numbers** first:

\[ 1101.11 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} \]

\[ = 8 + 4 + 1 + 0.5 + 0.25 \]

\[ = 13.75 \]
• If we know where the binary point is, we do not need to "store" it anywhere. (Remember we used a bit to represent the +/- sign in 2’s complement.)

• A format where the binary/decimal point is fixed between 2 groups of bits is called a **fixed-point format**.
Definition

- A number format where the numbers are **unsigned** and where we have \( a \) integer bits (on the left of the decimal point) and \( b \) fractional bits (on the right of the decimal point) is referred to as a \( U(a,b) \) fixed-point format.

- Value of an \( N \)-bit binary number in \( U(a,b) \):

\[
x = (1/2^b) \sum_{n=0}^{N-1} 2^n x_n
\]
Exercise 1

\[ x = 1011\ 1111 = 0xBF \]

- What is the value represented by \( x \) in \( U(4,4) \)?
- What is the value represented by \( x \) in \( U(7,3) \)?
Exercise 2

• \( z = 00000001\ 00000000 \)

• \( y = 00000010\ 00000000 \)

• \( v = 00000010\ 10000000 \)

• What values do \( z \), \( y \), and \( v \) represent in a \( U(8,8) \) format?
Exercise 3

• What is 12.25 in $U(4,4)$? In $U(8,8)$?
What about **Signed** Numbers?
Observation #1

• In an N-bit, unsigned integer format, the weight of the MSB is $2^{N-1}$
<table>
<thead>
<tr>
<th>nybble</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+0</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
</tr>
<tr>
<td>0111</td>
<td>+7</td>
</tr>
<tr>
<td>1000</td>
<td>+8</td>
</tr>
<tr>
<td>1001</td>
<td>+9</td>
</tr>
<tr>
<td>1010</td>
<td>+10</td>
</tr>
<tr>
<td>1011</td>
<td>+11</td>
</tr>
<tr>
<td>1100</td>
<td>+12</td>
</tr>
<tr>
<td>1101</td>
<td>+13</td>
</tr>
<tr>
<td>1110</td>
<td>+14</td>
</tr>
<tr>
<td>1111</td>
<td>+15</td>
</tr>
</tbody>
</table>

\[ N = 4 \]
\[ 2^{N-1} = 2^3 = 8 \]
Observation #2

• In an N-bit signed 2's complement, integer format, the weight of the MSB is $-2^{N-1}$
<table>
<thead>
<tr>
<th>nybble</th>
<th>2's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+0</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
</tr>
<tr>
<td>0111</td>
<td>+7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
</tr>
</tbody>
</table>

\[ N=4 \]
\[ -2^{N-1} = -2^3 = -8 \]
Fixed-Point Signed Format

- **Fixed-Point signed** format = sign bit + $a$ integer bits + $b$ fractional bits = $N$ bits = $A(a, b)$

- $N =$ number of bits = $1 + a + b$

- Format of an $N$-bit $A(a, b)$ number:

\[
x = (1/2^b) \left[ -2^{N-1}x_{N-1} + \sum_{0}^{N-2} 2^n x_n \right],
\]
Examples in A(7,8)

- 000000001 00000000 = 00000001 . 00000000 = ?
- 100000001 00000000 = 10000001 . 00000000 = ?
- 00000010 00000000 = 0000010 . 00000000 = ?
- 10000010 00000000 = 1000010 . 00000000 = ?
- 00000010 10000000 = 0000010 . 10000000 = ?
- 10000010 10000000 = 1000010 . 10000000 = ?
Examples in A(7,8)

- 000000001 00000000 = 00000001 . 00000000 = 1d
- 100000001 00000000 = 10000001 . 00000000 = -128 + 1 = -127d
- 00000010 00000000 = 0000010 . 00000000 = 2d
- 10000010 00000000 = 1000010 . 00000000 = -128 + 2 = -126d
- 00000010 10000000 = 0000010 . 10000000 = 2.5d
- 10000010 10000000 = 1000010 . 10000000 = -128 + 2.5 = -125.5d
Exercises

• What is -1 in $A(7,8)$?
• What is -1 in $A(3,4)$?
• What is 0 in $A(7,8)$?
• What is the smallest number one can represent in $A(7,8)$?
• The largest in $A(7,8)$?
Exercises

• What is the largest number representable in $U(a, b)$?

• What is the smallest number representable in $U(a, b)$?

• What is the largest positive number representable in $A(a, b)$?

• What is the smallest negative number representable in $A(a, b)$?
We Stopped Here Last Time...

http://i.imgur.com/doh3mlZ.jpg
• **Fixed-Point Format**
  
  • **Definitions**
    
    • Range
    
    • Precision
    
    • Accuracy
  
  • **Floating-Point Format**
Range

• Range = difference between most positive and most negative numbers.

• **Unsigned Range:**
The range of \( \text{U}(a, b) \) is \( 0 \leq x \leq 2^a - 2^{-b} \).

• **Signed Range:**
The range of \( \text{A}(a, b) \) is \( -2^a \leq x \leq 2^a - 2^{-b} \).
Precision

- **Precision** = \( b \), the number of fractional bits
  
  [https://en.wikibooks.org/wiki/Floating_Point/Fixed-Point_Numbers](https://en.wikibooks.org/wiki/Floating_Point/Fixed-Point_Numbers)

- **Precision** = \( N \), the total number of bits
  
Resolution

• The **resolution** is the smallest non-zero magnitude representable.

• The **resolution** is the size of the intervals between numbers represented by the format

• Example: \( A(13, 2) \) has a resolution of 0.25.
A(13, 2) $\rightarrow$ sbbbs bbbbs bbbbs bb . bb

\[ 2^{-2} = 0.25 \]
\[ 2^{-1} = 0.5 \]
A(13, 2) \rightarrow \text{sbbbb bbbbb bbbbb bb . bb}

Resolution

\[
2^{-2} = 0.25 \\
2^{-1} = 0.5
\]
Accuracy

• The **accuracy** is the largest magnitude of the difference between a number and its representation.

• **Accuracy** = $1/2$ **Resolution**
A(13, 2) \rightarrow \text{sbbbb bbbbb bbbbb bb . bb}

Real quantity we want to represent
A(13, 2) -> sbbb bbbb bbbb bb . bb

\[ 2^{-2} = 0.25 \]
\[ 2^{-1} = 0.5 \]

Real quantity we want to represent
A(13, 2) \rightarrow \ sbbb bbbb bbbb bb . bb

Error

2^{-2} = 0.25
2^{-1} = 0.5
$A(13, 2) \rightarrow \text{sbbbbb bbbbb bbbbb} \text{ bb . bb}$

Largest Error = **Accuracy**
Questions in search of answers…

• What is the accuracy of an U(7,8) number format?

• How good is U(7,8) at representing small numbers versus representing larger numbers? In other words, is the format treating small numbers better than large numbers, or the opposite?
Questions in search of answers…

• What is the accuracy of an U(7,8) number format?

• How good is U(7,8) at representing small numbers versus representing larger numbers? In other words, is the format treating small numbers better than large numbers, or the opposite?

Another way of thinking about this: Think of bits as colors, and the number as an amount of money paid for a portrait. The amount of money defines the number of colors used to paint.
• Fixed-Point Format

• Floating-Point Format
IEEE
Floating-Point Number Format
A bit of history…

• 1960s, 1970s: many different ways for computers to represent and process real numbers. Large variation in way real numbers were operated on

• 1976: Intel starts design of first hardware floating-point co-processor for 8086. Wants to define a standard

• 1977: Second meeting under umbrella of Institute for Electrical and Electronics Engineers (IEEE). Mostly microprocessor makers (IBM is observer)

• Intel first to put whole math library in a processor
# Intel Coprocessors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8087</td>
<td>1980</td>
<td>Numeric coprocessor for 8086 and 8088 processors.</td>
</tr>
<tr>
<td>80C187</td>
<td>19??</td>
<td>Math coprocessor for 80C186 embedded processors.</td>
</tr>
<tr>
<td>80287</td>
<td>1982</td>
<td>Math coprocessor for 80286 processors.</td>
</tr>
<tr>
<td>80387</td>
<td>1987</td>
<td>Math co-processor for 80386 processors.</td>
</tr>
<tr>
<td>80487</td>
<td>1991</td>
<td>Math co-processor for SX versions of 80486 processors.</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>2012</td>
<td>Multi-core co-processor for Xeon CPUs.</td>
</tr>
</tbody>
</table>
Integrated Coprocessor

(Early) Intel Pentium
Some Processors that do not contain FPU

- Some ARM processors
- Arduino Uno
- Others
How Much Slower is Library vs FPU operations?


Library-emulated FP operations = 10 to 100 times slower than hardware FP operations executed by FPU
Floating Point Numbers Are Weird…
“0.1 decimal does not exist”

— D.T.
6.02 \times 10^{23}
-0.0000001
1.23456789 \times 10^{-19}
-1.0
1.230

= 12.30 \times 10^{-1}

= 123.0 \times 10^{-2}

= 0.123 \times 10^{1}
IEEE Format

- 32 bits, single precision (floats in Java)
- 64 bits, double precision (doubles in Java)
- 80 bits*, extended precision (C, C++)

\[ x = +/- 1.\text{bbbbbb}...\text{bbb} \times 2^{\text{bbb}...\text{bb}} \]

*80 bits in assembly = 1 Tenbyte
Observations

\[ x = +/- 1.bbbbbbb....bbb \times 2^{bbb...bb} \]

- +/- is the sign. It is represented by a bit, equal to 0 if the number is positive, 1 if negative.

- the part 1.bbbbbbb....bbb is called the mantissa

- the part bbb...bb is called the exponent

- 2 is the base for the exponent (could be different!)

- the number is normalized so that its binary point is moved to the right of the leading 1.

- because the leading bit will always be 1, we don't need to store it. This bit will be an implied bit.
http://www.h-schmidt.net/FloatConverter/IEEE754.html
We stopped here last time...
for ( double d = 0; d != 0.3; d += 0.1 )
System.out.println( d );
Normalization  
(in decimal)  
(normal = standard form)  

\[ y = 123.456 \]  

\[ y = 1.23456 \times 10^2 \]
Normalization (in binary)

\[ y = 1000.100111 \quad (8.609375_{10}) \]

\[ y = 1.000100111 \times 2^{3} \]
Normalization (in binary)

\[ y = 1000.100111 \]

\[ y = 1.000100111 \times 2^3 \]
Normalization (in binary)

\[ y = 1000.100111 \]

\[ y = 1.000100111 \times 2^3 \]

\[ y = 1.000100111 \times 10^{11} \]
\[ +1.000100111 \times 10^{11} \]

- **Sign**: 0
- **Mantissa**: 1000100111
- **Exponent**: 11
But, remember, all* numbers have a leading 1, so, we can pack the bits even more efficiently!

*really?
\[
+1.000100111 \times 10^{11}
\]

```
0 0001001110 11
sign mantissa exponent
```
IEEE Format

24 bits stored in 23 bits!
$y = 1000.100111$

$y = 1.000100111 \times 2^3$

$y = 1.000100111 \times 10^{11}$
\[ y = 1.000100111 \times 10^{11} \]

Why not 00000011?
How is the exponent coded?
<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
<tr>
<td>real exponent</td>
<td>stored exponent</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>
\[ y = 1.000100111 \times 10^{11} \]

Ah! 3 represented by \( 130 = 128 + 2 \)

\[ 1.0761719 \times 2^3 = 8.6093752 \]
Verification
8.6093752 in IEEE FP?

http://www.h-schmidt.net/FloatConverter/IEEE754.html
Exercises

• How is 1.0 coded as a 32-bit floating point number?
• What about 0.5?
• 1.5?
• -1.5?
• what floating-point value is stored in the 32-bit number below?

1 | 1000 0011 | 111 1000 0000 0000 0000 0000 0000
what about 0.1?
0.1 decimal, in 32-bit precision, IEEE Format:

0 01111011 10011001100110011001101
0.1 decimal, in 32-bit precision, IEEE Format:

0 01111011 10011001100110011001101

Value in double-precision: 0.10000000149011612
NEVER
NEVER
NEVER
COMPARE FLOATS
OR DOUBLES FOR
EQUALITY!
N-E-V-E-R!
for ( double d = 0; d != 0.3; d += 0.1 )
System.out.println( d );
<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>
Zero

• Why is it special?

• $0.0 = 0 \ 00000000 \ 0000000000000000000000000000000000$
<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>
Very Small Numbers

- Smallest numbers have stored exponent of 0.
- In this case, the implied 1 is omitted, and the exponent is -126 (not -127!)
if mantissa is 0:
number = 0.0

if mantissa is !0:
no hidden 1

<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-127</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>-127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>
Very Small Numbers

- Example: $0 00000000 001000000000000000000000$

$$+ \left( 2^{-126} \right) \times \left( 0.001 \right)$$

$$+ \left( 2^{-126} \right) \times \left( 0.125 \right) = 1.469 \times 10^{-39}$$
<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>. . . .</td>
<td>. . . .</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>. . . .</td>
<td>. . . .</td>
<td></td>
</tr>
<tr>
<td>. . . .</td>
<td>. . . .</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>
Very large numbers

- stored exponent = 1111 1111
- if the mantissa is = 0 $\Rightarrow +/- \infty$
Very large numbers

• stored exponent = 1111 1111

• if the mantissa is = 0 $\rightarrow$ +/- $\infty$
Very large numbers

• stored exponent = 1111 1111

• if the mantissa is = 0 $\rightarrow$ +/- $\infty$

• if the mantissa is != 0 $\rightarrow$ NaN
Very large numbers

- stored exponent = 1111 1111

- if the mantissa is = 0 $\Rightarrow +/- \infty$

- if the mantissa is != 0 $\Rightarrow$ NaN = Not-a-Number
Very large numbers

- stored exponent = 1111 1111
- if the mantissa is = 0 ==> +/- \( \infty \)
- if the mantissa is != 0 ==> NaN
NaN is sticky!
• 0 11111111 000000000000000000000000 = + \infty

• 1 11111111 000000000000000000000000 = - \infty

• 0 11111111 100000100000000000000000 = NaN

- The **divisions** 0/0 and ±∞/±∞

- The **multiplications** 0×±∞ and ±∞×0

- The **additions** ∞ + (−∞), (−∞) + ∞ and equivalent subtractions

- The **square root** of a negative number.

- The **logarithm** of a negative number

- The **inverse sine or cosine** of a number that is less than −1 or greater than +1
public class GenerateNaN {
    public static void main(String args[]) {
        double[] allNaNs = { 0D / 0D,
            POSITIVE_INFINITY / POSITIVE_INFINITY,
            POSITIVE_INFINITY / NEGATIVE_INFINITY,
            NEGATIVE_INFINITY / POSITIVE_INFINITY,
            NEGATIVE_INFINITY / NEGATIVE_INFINITY,
            0 * POSITIVE_INFINITY,
            0 * NEGATIVE_INFINITY,
            Math.pow(1, POSITIVE_INFINITY),
            POSITIVE_INFINITY + NEGATIVE_INFINITY,
            NEGATIVE_INFINITY + POSITIVE_INFINITY,
            POSITIVE_INFINITY - POSITIVE_INFINITY,
            NEGATIVE_INFINITY - NEGATIVE_INFINITY,
            Math.sqrt(-1),
            Math.log(-1),
            Math.asin(-2),
            Math.acos(+2),
        };
        System.out.println(Arrays.toString(allNaNs));
        System.out.println(NaN == NaN);
        System.out.println(Double.isNaN(NaN));
    }
}
Range of Floating-Point Numbers

<table>
<thead>
<tr>
<th></th>
<th>Denormalized</th>
<th>Normalized</th>
<th>Approximate Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>± $2^{-149}$ to $(1-2^{-23})\times2^{-126}$</td>
<td>± $2^{-126}$ to $(2-2^{-23})\times2^{127}$</td>
<td>± $\sim10^{-44.85}$ to $\sim10^{38.53}$</td>
</tr>
<tr>
<td>Double Precision</td>
<td>± $2^{-1074}$ to $(1-2^{-52})\times2^{-1022}$</td>
<td>± $2^{-1022}$ to $(2-2^{-52})\times2^{1023}$</td>
<td>± $\sim10^{-323.3}$ to $\sim10^{308.3}$</td>
</tr>
</tbody>
</table>
Range of Floating-Point Numbers

Remember that!

<table>
<thead>
<tr>
<th></th>
<th>Denormalized</th>
<th>Normalized</th>
<th>Approximate Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Precision</strong></td>
<td>$\pm 2^{-149}$ to $(1-2^{-23}) \times 2^{-126}$</td>
<td>$\pm 2^{-126}$ to $(2-2^{-23}) \times 2^{127}$</td>
<td>$\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>$\pm 2^{-1074}$ to $(1-2^{-52}) \times 2^{-1022}$</td>
<td>$\pm 2^{-1022}$ to $(2-2^{-52}) \times 2^{1023}$</td>
<td>$\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th><strong>Binary</strong></th>
<th><strong>Decimal</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Precision</strong></td>
<td>$\pm (2-2^{-23}) \times 2^{127}$</td>
<td>$\sim \pm 10^{38.53}$</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>$\pm (2-2^{-52}) \times 2^{1023}$</td>
<td>$\sim \pm 10^{308.25}$</td>
</tr>
</tbody>
</table>
Resolution of a Floating-Point Format

Check out table here: http://tinyurl.com/FPResol
Resolution
Another way to look at it

http://jasss.soc.surrey.ac.uk/9/4/4.html
• Rosetta Landing on Comet

• 10-year trajectory
Why not using 2’s *Complement* for the Exponent?

0.000000005 = 0 01100110 1010110101111111110010101
1 = 0 01111111 00000000000000000000000000000000
65536.5 = 0 10001111 000000000000000000000001000000
65536.25 = 0 10001111 000000000000000000000001000000
We stopped here last time...
END OF THE SEMESTER!
Does this converter support NaN, and \( \infty \)?

Are there several different representations of \( +\infty \)?

What is the largest float representable with the 32-bit format?

What is the smallest normalized float (i.e. a float which has an implied leading 1. bit)?
How do we add 2 FP numbers?
• $fp_1 = s_1 \ m_1 \ e_1$
  $fp_2 = s_2 \ m_2 \ e_2$
  $fp_1 + fp_2 = ?$

• **denormalize** both numbers (restore hidden 1)

• assume $fp_1$ has largest exponent $e_1$: make $e_2$ **equal** to $e_1$ and **shift decimal point** in $m_2 \rightarrow m_2'$

• compute **sum** $m_1 + m_2'$

• **truncate** & **round** result

• **renormalize** result (after checking for special cases)
\[ 1.111 \times 2^5 + 1.110 \times 2^8 \]

\[
\begin{align*}
&1.11000000 \times 2^8 \\
+ &0.00111100 \times 2^8 \\
\hline
&1.11111100 \times 2^8 \\
\end{align*}
\]

= 10.000 \times 2^8

= 1.000 \times 2^9
How do we multiply 2 FP numbers?
• $fp1 = s_1 m_1 e_1$
  $fp2 = s_2 m_2 e_2$
  $fp1 \times fp2 = ?$

• Test for multiplication by special numbers (0, NaN, $\infty$)

• **denormalize** both numbers (restore hidden 1)

• compute product of $m_1 \times m_2$

• compute **sum** $e_1 + e_2$

• **truncat**e & **round** $m_1 \times m_2$

• adjust $e_1 + e_2$ and **normalize.**
How do we compare two FP numbers?
As unsigned integers!
No unpacking necessary!
Programming FP Operations in Assembly...
Pentium

- EAX
- EBX
- ECX
- EDX

ALU
Cannot do FP computation
Intel Pentium 5 Prescott

Instruction Trace Cache
- 16k uOps
- 128 kByte
- 8 way set associative
- 8 x 512 sets of 4 uOps
- Tag comparators
- 39 bit virtual Tag
- Misc. Tag Data

Execution Pipeline Start
- Micro code Sequencer
- Flash & ROM
- 16k uOps
- 128 kByte
- 8 way set associative
- 8 x 512 sets of 4 uOps
- Tag comparators
- 39 bit virtual Tag
- Misc. Tag Data

Buffer Allocation & Register Rename
- Instruction Queue (for less critical fields of the uOps)
- General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)
- uOp Schedulers
  - Parallel (Matrix) Scheduler for the two double pumped ALU's
  - General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)
  - FP Move Scheduler: (8x8 dependency matrix)
  - Load / Store Linear Address Collision History Table
  - Load / Store uOp Scheduler: (8x8 dependency matrix)

FP, MMX, SSE1..3
- Floating Point Registers
- Legacy Floating Point
- Floating Point Multiply
- Legacy Floating Point Add

Integer Execution Core
- uOp Dispatch unit & Replay Buffer: Dispatches up to 6 uOps / cycle
- Integer Renamed Register File: 256 entries of 32 bit (+ 6 status flags)
  - 12 read ports and six write ports
- Databus switch & Bypasses to and from the Integer Register File.
- Flags, Write Back
- Double Pumped ALU 0
- Double Pumped ALU 1
- Load Address Generator Unit
- Store Address Generator Unit
- Load Buffer (96 entries)
- Store Buffer (48 entries)

Instruction Fetch from L2 cache and Branch Prediction
- 512 kByte L2 Cache Block
- L2 Phys. Tags
- L2 Cache Line Transfer Buffers

Front Side Bus Interface, 533..800 MHz

April 19, 2003
www.chip-architect.com

Operation: \((7+10)/9\)
Operation: \((7+10)/9\)

fpush 7
Operation: $(7+10)/9$

- `fpush 7`
- `fpush 10`
Operation: \((7+10)/9\)

fpush 7
fpush 10
fadd
Operation: \((7+10)/9\)

- `fpush 7`
- `fpush 10`
- `fadd`
Operation: \((7+10)/9\)

```
fpush 7
fpush 10
fadd
fpush 9
```
Operation: \((7+10)/9\)

- `fpush 7`
- `fpush 10`
- `fadd`
- `fpush 9`
- `fdiv`

<table>
<thead>
<tr>
<th>SP0</th>
<th>SP1</th>
<th>SP2</th>
<th>SP3</th>
<th>SP4</th>
<th>SP5</th>
<th>SP6</th>
<th>SP7</th>
</tr>
</thead>
</table>

Floating Point Unit
Operation: \((7+10)/9\)

- `fpush 7`
- `fpush 10`
- `fadd`
- `fpush 9`
- `fdiv`
The Pentium computes FP expressions using RPN!
The Pentium computes FP expressions using RPN!
Nasm Example: \( z = x + y \)

```assembly
SECTION .data

x dd 1.5
y dd 2.5
z dd 0

; compute \( z = x + y \)
SECTION .text

fld dword [x]
fld dword [y]
fadd
fstp dword [z]
```
Printing floats in C

```c
#include "stdio.h"

int main() {
    float z = 1.2345e10;
    printf("z = %e\n\n", z);
    return 0;
}
```
Printing floats in C

```c
#include "stdio.h"

int main() {
    float z = 1.2345e10;
    printf("z = %e\n\n", z);
    return 0;
}
```

```
gcc -m32 -o printFloat printFloat.c
./printFloat
z = 1.234500e+10
```
Printing floats in Assembly?

```
asm program
call printf
```

```
C stdio.h library (printf)
```

```
nasm
object file
```

```
gcc executable
```
extern printf ; the C function to be called

SECTION .data ; Data section
msg db "sum = %e",0x0a,0x00
x dd 1.5
y dd 2.5
z dd 0
temp dq 0

SECTION .text ; Code section.
global main ; "C" main program
main:
    ; label, start of main program
    ; need to convert 32-bit to 64-bit
    fld dword [x]
    fld dword [y]
fadd
    fstp dword [z] ; store sum in z
    fld dword [z] ; transform z to 64-bit by pushing in stack
    fstp qword [temp] ; and popping it back as 64-bit quadword
    push dword [temp+4] ; push temp as 2 32-bit words
    push dword [temp]
    push dword msg ; address of format string
call printf ; Call C function
    add esp, 12 ; pop stack 3*4 bytes
    mov eax, 1 ; exit code, 0=normal
    mov ebx, 0
    int 0x80 ;
dthiebaut@hadoop:~/temp$ nasm -f elf addFloats.asm
dthiebaut@hadoop:~/temp$ gcc -m32 -o addFloats addFloats.o

dthiebaut@hadoop:~/temp$ ./addFloats
sum = 4.000000e+00

dthiebaut@hadoop:~/temp$
More code examples here:

http://cs.smith.edu/dftwiki/index.php/CSC231_An_Introduction_to_Fixed-and_Floating-Point_Numbers#Assembly_Language_Programs